TITLE:

Validation of Numerical Computation

AUTHOR(S):

Rall, L.B.

CITATION:

Rall, L.B.. Validation of Numerical Computation. 数理解析研究所講究録 1988, 673: 1-16

ISSUE DATE:

1988-11

URL:

http://hdl.handle.net/2433/100908

RIGHT:

京都大学学術情報リポジトリ KURENAI 紅
Kyoto University Research Information Repository

数理解析研究所講究録
第673巻 1988年 1-16

Validation of Numerical Computation

L. B. Rall

Center for the Mathematical Sciences
University of Wisconsin-Madison
610 Walnut Street
Madison, Wisconsin 53705
U.S.A.

1. The problem of validation. Mathematics has traditionally set very high standards for validity of results. Axioms and definitions are framed, and conclusions of theorems are shown to follow by rigorous deductive reasoning. Before the advent of the digital computer, much the same standards were applied to numerical calculations such as the production of reliable tables of mathematical quantities for general use. Unfortunately, standards seem to have been lowered considerably, and now numerical results are often accepted with little or no guarantee of validity. Frequently, correct programming of the algorithm and perhaps the use of double-precision arithmetic are considered adequate, even though counter-examples abound. Other validation heuristics are based on familiarity with the problem and algorithms for its solution, but this level of sophistication is not always present among users of a given program. For some important problems, of course, there are detailed error analyses which can be used to validate computed results, but generally validation as mathematicians know it is left at the door of the computing laboratory.

A very invidious present trend is to value speed of computation above all, even though it is obvious that *an incorrect result is still* wrong, *no matter how rapidly it is computed.* The same is true for results of dubious validity, and time spent on investigating their accuracy nullifies any advantage of rapid computation.

The uncritical acceptance of unvalidated results is not only unsatisfying intellectually, it is courting disaster. An alternative is to carry the validation process used in the mathematical formulation and solution of the problem through to the final numerical results. This means that a definition of what is meant by validity of admittedly approximate numerical results has to be framed. One workable definition is:

Definition. *To be valid, a computation should provide a guarantee of the accuracy of its results, or a statement that results of guaranteed accuracy could not be obtained.*

In the case of failure, some reasons pointing to limitations of the method or data used would also be helpful.

The problem of validation is very complicated, which may be one reason that it has not been pursued with the zeal it deserves. At first glance, one is put off by the kind of tedious roundoff error analysis which may depend on the idiosyncracies of the machine being used. However, the true problem is at a deeper level. Validation really depends on interaction between the mathematical algorithms chosen, the software used, and the arithmetics of the computer (and thus ultimately the hardware) [13]. The basic ideas can be summarized as follows:

(a) The computer should be constructed so that its arithmetics satisfy axioms, and valid conclusions can be drawn about the results of operations.

(b) Programming languages used should facilitate clear and correct implementation of algorithms, and additionally give the programmer easy access to special features of the computer arithmetics, such as directed rounding or accurate scalar products of vectors.

(c) Mathematical algorithms for the solution of a problem should be selected with validation in mind, that is, the computer should do as much of the work as possible to validate results as well as calculate them.

These points will be discussed in more detail below. First, an example will be given to illustrate the concept of validation more precisely.

2. Example: Solution of Linear Systems. The solution of a linear system $Ax = b$ of n equations in n unknowns has a long history in computational mathematics. Over the years, a *standard method* has evolved, based on Gaussian elimination. This method has been well programmed and tested and is widely available, for example, in PC-MATLAB

[14]. The standard method gives as its result a vector of floating-point numbers which very often is a good approximation to the solution x. A *validated method* for solution of systems, due to S. M. Rump ([13], pp. 53–120) requires an accurate scalar product and directed rounding (or interval arithmetic), and is available in Pascal-SC [3], [11]. The validated method either gives a matrix of floating-point intervals which contain the exact values of the solution x (in this case the algorithm verifies the existence of the solution), or else a message that no such interval vector could be obtained.

The standard and validated methods were applied to the linear system of eight equations in eight unknowns with matrix $A = [a_{ij}]$, where

$$a_{ij} = \frac{(-1)^{i+j}(i+8)!(j+8)!}{(i+j-1)[(i-1)!(j-1)!]^2(9-i)!(9-j)!}, \tag{2.1}$$

$i, j = 1, 2, \ldots, 8$. The matrix A has integer components, and is the inverse of the notorious Hilbert matrix $H = [h_{ij}]$ of order 8 with components

$$h_{ij} = \frac{1}{i+j-1}, \qquad i, j = 1, 2, \ldots, 8, \tag{2.2}$$

[5]. The components of A can be represented exactly by floating-point numbers with mantissas of 10 decimal digits.

The right-hand side b was taken to be the unit vector

$$b = (1, 0, 0, 0, 0, 0, 0, 0), \tag{2.3}$$

so the correct solution is the first column of the Hilbert matrix (2.2):

$$x = \left(1, \frac{1}{2}, \frac{1}{3}, \frac{1}{4}, \frac{1}{5}, \frac{1}{6}, \frac{1}{7}, \frac{1}{8}\right). \tag{2.4}$$

Solution of the system $Ax = b$ by both methods give the following results for x:

Standard Method	Validated Method
0.56249998880618	[0.999999999999, 1.00000000001]
0.14999999105731	[0.499999999999, 0.500000000001]
0.04166665922158	[0.333333333333, 0.333333333334]
−0.00000000637690	[0.249999999999, 0.250000000001]
−0.01875000557662	[0.199999999999, 0.200000000001]
−0.02777778273250	[0.166666666666, 0.166666666667]
−0.03214286160041	[0.142857142857, 0.142857142858]
−0.03409091314193	[0.124999999999, 0.125000000001]

Here, a usually reliable method fails to compute a single correct significant digit of the answer, and even the signs of most of the components of its result are wrong. Furthermore, and perhaps even more importantly, there was no warning whatever of this lack of validity. The results of the validated method, by contrast, are of an entirely different quality. They actually verify that A is invertible, and are *interval inclusions* of the *exact* solution x of the system ([13], pp. 53–120). Furthermore, in 12 decimal digit floating-point arithmetic, the inclusions given by the validated method are of *maximum quality*, meaning that they are the smallest floating-point intervals which contain the exact results in their interior. It might be mentioned that the completely wrong results computed by the standard method were obtained *very rapidly*, using the vaunted 8087 mathematics coprocessor. The valid results from Pascal-SC used a software implementation of the floating-point arithmetic of Kulisch and Miranker [12], so the computation was much slower. Of course, those who value speed above all are welcome to wrong answers, but there is no essential conflict between speed and accuracy. Once a method for computation of trustworthy results has been developed, it is usually just a technological problem to increase its speed without loss of validity.

3. Computer arithmetics and roundings. Computer arithmetics are referred to in the plural because a careful analysis shows that there are actually many types of arithmetic being done by the computer [12]. First, there are the logical (Boolean) and integer arithmetics. Both will be assumed to be done correctly by the computer, the latter within the range determined by the computer architecture. In fact, all computer arithmetics deal with a finite set of values, so subsequent remarks will be assumed to apply to results quantities in the ranges so determined.

The main difficulty occurs in the approximation of the real numbers $\mathbf{R}$ by a finite subset $\mathbf{S}$ consisting of the floating-point numbers with which a given computer works. This is really the basic problem, because approximation of complex numbers, intervals, vectors, matrices, and so on can be considered in terms of the componentwise approximation of ordered arrays of real numbers by corresponding arrays of floating-point numbers. This leads directly to the question of a mapping, or rounding, $\Box : \mathbf{R} \to \mathbf{S}$ from the real numbers $\mathbf{R}$ to the corresponding grid $\mathbf{S}$ of floating-point numbers. This problem is considered in detail by Kulisch and Miranker [12], who formulate the following axioms:

$$(3.1) \qquad \bigwedge_{u \in \mathbf{S}} \Box u = u, \qquad \bigwedge_{r,s \in \mathbf{R}} r \leq s \Rightarrow \Box r \leq \Box s, \qquad \bigwedge_{r \in \mathbf{R}} \Box(-r) = -\Box r.$$

It follows immediately that if u, v with $u < v$ are *adjacent* floating-point numbers, that is, there exists no floating-point number w such that $u < w < v$, and r is a real number such that $u \leq r \leq v$, then $\square r = u$ or $\square r = v$. In either case, there is no floating-point number between r and $\square r$, and the approximation of r by $\square r$ will be said to be of *maximum quality*. Various actual roundings can implement $\square$, such as rounding to the closer of u, v (with a tie-breaking rule satisfying the third axiom), or rounding to or away from 0 [12].

Note that the *precision*, or number of digits used in the representation of floating-point numbers, does not enter into this discussion of rounding. The precision will determine the spacing between adjacent floating-point numbers, and hence the accuracy of results of maximum quality.

The rounding operator determines the properties of the floating-point arithmetic operators. If $\boxplus$ denotes the floating-point operator corresponding to the arithmetic operator $\star \in \{+, -, \cdot, /\}$, then it is required that

$$\bigwedge_{u,v \in \mathbf{S}} u \boxplus v = \square (u \star v). \tag{3.2}$$

For interval arithmetic, the *directed roundings* ∇, Δ of maximum quality such that

$$\bigwedge_{r \in \mathbf{R}} \nabla r \leq r, \quad \bigwedge_{r \in \mathbf{R}} \Delta r \geq r, \tag{3.3}$$

are also needed. For intervals with real endpoints $r \leq s$, $\square [r, s] = [\nabla r, \Delta s]$. This increases the number of basic arithmetic operations to twelve.

Interval arithmetic is, of course, the fundamental tool for validation of numerical results [1], [6], [7], [12], [13], [15]–[27]. While a real number $r \notin \mathbf{S}$ cannot in be represented exactly by a floating-point number, it will be contained in an interval $[u, v]$ with endpoints which are adjacent floating-point numbers. Thus, the computation of a result r is said to be *validated* if the algorithm verifies the existence of r in a floating-point interval $[a, b]$. The *quality* of this inclusion depends on the width of $[a, b]$. If $a < b$ and the open interval (a, b) contains at most one floating-point number w, then an inclusion of *maximum quality* has been obtained, as in the matrix inversion in §2. In the case (a, b) contains no floating-point numbers, then either a or b is an approximation of maximum quality to r, otherwise, the interior point w is. An inclusion of maximum quality, is, of course, the ideal solution to the problem of validation of a numerical computation. Otherwise, the goal is to produce an interval inclusion of as small width as possible, or one which is satisfactory for the problem being solved.

In addition to the real numbers, scientific computation works with structured types, which are generally represented as elements of Cartesian products of real numbers, for example, an ordered pair (a, b) can represent a complex number. However, the rules of arithmetic and the corresponding rounded arithmetic operators will be distinct for each such type. The axioms of rounding are applied componentwise to structured types, and the corresponding floating-point operators are defined accordingly. For example, for complex arithmetic, the rules are:

$$
\begin{aligned}
&(a,b) \pm (c,d) = (a \pm c, b \pm d), \\
&(a,b) \cdot (c,d) = (ac - bd, ad + bc), \\
&(a,b)/(c,d) = \left(\frac{ac + bd}{c^2 + d^2}, \frac{bc - ad}{c^2 + d^2} \right), \qquad (c,d) \neq (0,0).
\end{aligned} \tag{3.4}
$$

It is obvious that special algorithms must be used for multiplication and division of complex numbers, since, for example

$$\square (ac - bd) \neq (a \boxdot c) \boxminus (b \boxdot c) \tag{3.5}$$

in general. Thus, simulation of complex arithmetic by real floating-point arithmetic is of inadequate quality. Complex interval arithmetic can be defined componentwise as in the case of real interval arithmetic [12].

An important consequence of the work of Kulisch and Miranker [12] is that maximum quality real vector and matrix arithmetic requires the computation of the scalar product of floating-point vectors u, v to be of maximum quality, that is,

$$\bigwedge_{u,v \in \mathbf{S}^n} u \boxdot v = \square (u \cdot v) = \square \left(\sum_{i=1}^{n} u_i v_i \right). \tag{3.6}$$

This operation turns out to be useful to construct the operators needed to implement other computer arithmetics, such as complex arithmetic. Directed rounding of the scalar product (3.6) is also required to implement floating-point arithmetic for interval vectors and matrices. The scalar product (3.6) is also required for complex vector and matrix arithmetic, and, with directed rounding, for complex interval vector and matrix arithmetic.

So far, ten basic computer arithmetics and many corresponding operators have been introduced. Even more operators are defined if interactions between types are taken into account, for example, multiplication of vectors by scalars, addition of integers to complex numbers, etc. There are also other arithmetics which can be useful in the solution of

various problems, and can be implemented in similar ways to the ones discussed. For purposes of validation, some *differentiation arithmetics* and their interval counterparts have proved to be highly useful. In the simplest case of differentiation arithmetic, the pair (u, u') represents the value $u = u(x)$ of a function and the value $u' = u'(x)$ of its derivative at some point x ([9], pp. 287–295, [25]). Arithmetic operations for this type are defined by

$$
\begin{aligned}
&(u,u') \pm (v,v') = (u+v, u'+v'),\\
&(u,u') \cdot (v,v') = (uv, uv'+vu'),\\
&(u,u')/(v,v') = \left(\frac{u}{v}, \frac{vu'-uv'}{v^2}\right), \qquad v \neq 0.
\end{aligned} \tag{3.7}
$$

Differentiation arithmetic, starting with the pair $(x, 1)$, can be used to calculate the pair $(f(x), f'(x))$ for rational functions f *without* symbolic differentiation. This arithmetic is somewhat simpler than complex arithmetic, but again special algorithms are needed for floating-point multiplication and division of maximum quality ([9], pp. 287–295). The corresponding interval differentiation arithmetic is obtained if u, u', v, v' are taken to be intervals, and interval arithmetic is used. In this case, for rational f and $X = [a, b]$, the initial pair $(X, [1, 1])$ gives $(F(X), F'(X))$ as the result of the interval evaluation of f, where $f(X) \subseteq F(X)$ and $f'(X) \subseteq F'(X)$, in other words, $F(X)$ includes all possible values of $f(x)$ and $F'(X)$ includes all possible values of $f'(x)$ on $a \leq x \leq b$.

Differentiation arithmetic can be extended to higher dimensions and derivatives ([13], pp. 291–309, [25]). For example, for differentiable functions of n variables, variables of *type gradient* are represented by ordered pairs

$$
u = (u(x), u'(x)), \qquad x = (x_1, x_2, \ldots, x_n), \tag{3.8}
$$

where the first component is the value of u at x (a scalar), and $u'(x)$ is the gradient vector

$$
u'(x) = \left(\frac{\partial u}{\partial x_1}, \frac{\partial u}{\partial x_2}, \ldots, \frac{\partial u}{\partial x_n}\right) \tag{3.9}
$$

of u at x ([13], pp. 291–309). As in the case of the basic derivative type, the rules of arithmetic for gradient variables are given by (3.7), and incorporate the corresponding rules for differentiation of the corresponding operator. Thus, use of this arithmetic gives values of functions and their gradient vectors without symbolics or approximations. The

computer arithmetic for type gradient is based on the rounding operator $\square$ according to (3.2). Similarly, *Hessian* variables are ordered triples

$$u = (u(x), u'(x), u''(x)), \tag{3.10}$$

where $u''(x)$ is the Hessian matrix

$$u''(x) = \left[\frac{\partial^2 u(x)}{\partial x_i \partial x_j} \right]. \tag{3.11}$$

The corresponding interval gradient and interval Hessian variables give guaranteed inclusions of the function values, gradient vectors, and, in the case of the latter, Hessian matrices as in the case of the basic differentiation and interval differentiation types. Real and interval gradient and Hessian variables are useful in the validated solution of a number of problems, such as systems of nonlinear equations [17], [24] and nonlinear optimization [18].

Other useful differentiation arithmetics are based on *Taylor* variables, which in the real case are $n+1$-tuples

$$u = (u_0, u_1, \ldots, u_n), \tag{3.12}$$

where each component is the value of the corresponding Taylor coefficient

$$u_k = \frac{1}{k!} u^{(k)}(x) h^k, \qquad k = 0, 1, \ldots, n, \tag{3.13}$$

in the expansion of $u(x+h)$. Once again, the rules of arithmetic for Taylor series are used to construct the corresponding computer arithmetic [25], and no symbolic differentiation is required. The corresponding interval Taylor type gives inclusions of the Taylor coefficients of u over an interval X with step H which is also interval-valued in general. R. E. Moore has shown the usefulness of these types for validated solution of ordinary differential equations, and numerical integration [15], [16], [22], [23].

Thus, there are a wide variety of computer arithmetics, each with a corresponding set of arithmetic operators. While such operators for structured types can be simulated by real or interval floating-point arithmetic (the so-called *vertical* definition of a computer arithmetic [13]), maximum quality requires special algorithms satisfying the condition (3.2). Of course, having individual operators of maximum quality is only a step toward validation, since the result of several operations could be of lower quality. Thus, the algorithms being used will figure significantly into the validation process.

4. Programming languages. The discussion of computer arithmetics in the previous section indicates some of the properties which a suitable programming language should have. A main purpose of such a language is to facilitate convenient, clear, and correct programming of mathematical algorithms, as well as validation of their results. In essence, the programming language is what transforms mathematical algorithms into computer arithmetics, and so it should have close connections to both. Unfortunately, most programming languages in common use at the present time fail to attain these goals. A typical bad example is ordinary Pascal, which does not specify any standards for floating-point arithmetic at all. Furthermore, suppose one wants to evaluate the formula

$$w = \sin\left(\frac{az+b}{cz+d}\right), \tag{4.1}$$

approximately, where a, b, c, d, z and consequently w are complex numbers. In Pascal, it is easy to define a data type `complex` to represent complex numbers, for example, by

```
type complex = record re,im:  real end;
```

However, no arithmetic operators can be defined for this type, and Pascal functions can only yield results of simple type, not structured types such as complex. Thus, approximate evaluation of (4.1) would have to be done by a sequence of procedure calls such as:

```
cmul(a,z,num);
cadd(num,b,num);
cmul(c,z,denom);
cadd(denom,d,denom);
cdiv(num,denom,quo);
csin(quo,w);
```

On the other hand, the same computation would be programmed as

```
w := csin((a*z+b)/(c*z+d));
```

in Pascal-SC, which also imposes standards of quality on computer arithmetics [3]. The relationship of this line of code to (4.1) is much easier to interpret than the corresponding sequence of procedure calls of ordinary Pascal, a language which is out of touch with computer arithemetics on one hand and mathematical algorithms on the other. Other popular languages exhibit similar incongruities.

With the above considerations in mind, it is possible to formulate some basic requirements for a satisfactory language:

(a) Type and operator concepts for definition and execution of computer arithmetics, and standards for quality.

(b) Functions which return results of appropriate type.

(c) A library of standard functions and utility routines which return results of maximum or high quality for the fundamental arithmetics.

With regard to (a), programmers should be able to specify that certain variables are real, interval, complex vectors, and so on, depending on the problem. The language should have an operator concept, so addition of two variables can be written as $a+b$, and will be carried out if addition is actually defined for whatever types of variables a, b happen to be. In other words, a notation similar to ordinary mathematical formulas can be used in writing a program. This is a convenience not only in programming, but also for understanding what a program written by someone else actually does. Languages in which the most basic operations on structured types have to be done by calling subroutines often lead to the production of bewildering programs for even simple tasks.

The set of operator symbols in the language should also be large enough to specify special characteristics of the arithmetic being used. For example, if directed rounding is available for the operation $\star$, then operator symbols for $\triangledown$ and $\triangle$ should be available in addition to the one for $\boxplus$. The programmer should also be able to define operators for special purposes, either by "overloading" existing operator symbols to apply to new types, or else by introducing operator identifiers for this purpose.

As specified by (b), functions defined in a given language should be able to return results of appropriate type. For example, the logarithmic function $\ln(x)$ is defined for real, complex, interval, and derivative variables, and should return the appropriate value depending on the type of x. (For derivative variables, functions are defined by the chain rule or the corresponding recurrence relations, for example,

$$\ln(u, u') = \left(\ln u, \frac{u'}{u}\right), \tag{4.2}$$

for the basic derivative or gradient type [25].)

Most programming languages meet the requirement (c) in one way or another, for the most commonly used functions, but leave the question of quality open. The ideal

situation would be to compute $\square f(x)$ for each standard function f, that is, the returned result would be of maximum quality. This can be done in most cases, but could be very expensive if $f(x)$ is actually close to a floating-point number w. In this situation, a result of *high quality* is returned as one of the endpoints u, v of the floating-point interval $[u, v]$ containing only w in its interior. Here, at most one floating-point number lies between the computed value and the actual result. Utility subroutines, such as solution of linear systems or polynomial equations, should conform to the same kind of standards.

The existing computer language which conforms most closely to these requirements, including standards of quality for floating-point arithmetics, is Pascal-SC [3]. This is because Pascal-SC was in fact developed to implement the Kulisch/Miranker theory of computer arithmetics [12], [13]. Ada has similar capabilities, and in addition requires maximum quality of the four basic arithmetic operations for real floating-point numbers. Most other languages seem to ignore the issue of standards for arithmetics, and provide only a meager set of operators for a few basic types, leaving other computer arithmetics to be simulated by real floating-point arithmetic, with a consequent loss of quality and increased difficulty of validation.

5. Algorithms for validation. In addition to computer arithmetics and programming languages with the properties described above, the validation process requires the selection of mathematical algorithms which produce results which can be computationally verified to be valid. Since the key idea is to bound the exact result by computable quantities, the mathematical foundations of such algortihms hark back to the work of Birkhoff on lattice theory [2] and Kantorovich on functional analysis in partially ordered spaces [8]. An example of early work on validated computation is the theory of inverse monotone operators, developed by Collatz and his school to find lower and upper bounds for the solutions of differential and other operator equations [4]. In particular, algorithms should be chosen to have the following properties:

(a) The existence of the result can be computationally verified.

(b) An interval inclusion of the result can be obtained which is of sufficiently small width, with high or maximum quality of the inclusion being the ultimate goal.

A number of algorithms of the type specified here have already been developed for purposes such as solution of linear equations and matrix inversion, evaluation and finding roots of polynomials, calculation of eigenvalues and eigenvectors, numerical integration, and so on

[9], [13]. To date, success has usually been obtained by one of two approaches. In the first case, the algorithms used are based on an iteration which has contractive properties, so that existence follows from the Banach or Schauder theorems [10], and the result is enclosed in a sequence of smaller and smaller intervals. In the second instance, the result is known to be the sum of a quantity which can be computed accurately and an unknown truncation error which can be enclosed by an interval which can be made arbitrarily narrow. (The existence of the interval inclusion of the truncation error implies the existence of the result in this case.) These types of algorithms will be illustrated by an example of each.

If the problem to be solved can be transformed into a fixed point problem

$$x = f(x) \tag{5.1}$$

for continuous f, the existence of a fixed point in an interval X can be investigated by use of an interval inclusion $F(X)$ of f on X. For example if

$$F(X) \subseteq X, \tag{5.2}$$

then the existence of a fixed point of f in X is verified on the basis of the Schauder theorem [10], since intervals are closed, convex, and compact. On the other hand, if

$$X \cap F(X) = \emptyset, \tag{5.3}$$

then f cannot have a fixed point in X, so a computational verification of *nonexistence* is also possible. In case of existence, the width of the interval inclusion of the fixed point x is reduced by *interval iteration* [15], [16]

$$X_{k+1} = X_k \cap F(X_k), \quad X_0 = X, \tag{5.4}$$

until the interval inclusion is of minimal or satisfactory width. By contrast, ordinary floating-point iteration cannot establish existence or nonexistence of solutions, even if apparently convergent or divergent.

If the problem is formulated as $g(x) = 0$ for differentiable g, then a useful interval inclusion of a solution is the one due to Krawczyk [16], [19], [20], [21]. This makes use of the fact that a zero of g is a fixed point of f defined by $f(x) = x - \Gamma^{-1}g(x)$, where Γ is a nonsingular matrix. The interval mean-value theorem gives the interval inclusion

$$F(X) = m(X) - \Gamma^{-1}g(m(X)) + \Gamma^{-1}\big(\Gamma - G'(X)\big)\big(X - m(X)\big), \tag{5.5}$$

of f, where $m(X)$ denotes the midpoint of X and $G'(X)$ is calculated by interval differentiation arithmetic. The interval $F(X)$ will contain any fixed points of f in X, so the conclusions of (5.2) or (5.3) hold for zeros of g in X. Since the unit ball $\{x : \|x\|_\infty = 1\}$ in $\mathbf{R}^n$ can be identified with the interval vector with all components equal to $[-1, 1]$, the definitions of the supremum vector and matrix norms can be carried over to interval transformations. In particular, for the choice

$$\Gamma = m(G'(X)), \tag{5.6}$$

one has

$$w(F(X)) \leq \Delta \left(w\big(m(X) - \Gamma^{-1} g(m(X)\big) + \|I - \Gamma^{-1} G'(X)\|_\infty w(X)\right). \tag{5.7}$$

As $w(X) \to 0$, it follows that $w(F(X))$ is limited only by the quality with which the various quantities involved can be computed. In the case of linear equations, $g(x) = Ax - b$, then $G'(X) = A$, and taking Γ^{-1} to be an approximate inverse of A, (5.7) becomes the interval residual correction formula used to validate the system solution in §2 ([13], pp. 53–120).

As the final example, consider a numerical integration formula of order k on n points,

$$\int_a^b f(x)dx = \sum_{i=1}^{n} w_i f(x_i) + C^{n,k} \frac{1}{(k+1)!} f^{(k+1)}(\xi) h^{k+1}, \tag{5.8}$$

where $h = O(1/n)$ and the point ξ in the interval $X = [a, b]$ of integration is unknown. Obviously,

$$\int_a^b f(x)dx \in \sum_{i=1}^{n} w_i f(x_i) + C^{n,k} \frac{1}{(k+1)!} F^{(k+1)}(X) h^{k+1}, \tag{5.9}$$

where interval Taylor arithmetic is used to calculate an inclusion of the trunction error. This calculation verifies that f is $k+1$-times differentiable on $[a, b]$ in addition to the inclusion (5.9), so the validity of formula (5.8) is established computationally. For the standard types of integration formulas, the width of the error term can be made small enough so the width of the inclusion of the integral depends on the quality with which the summation can be calculated. To make the width of the inclusion as small as possible, (5.9) is computed as the maximum quality scalar product of the interval vector

$$F = \big(f(x_1), f(x_2), \ldots, f(x_n), F_{k+1}(X, h)\big), \tag{5.10}$$

depending only on the integrand, where the last term is the interval Taylor coefficient of order $k+1$ of f, and the interval vector

$$W = (w_1, w_2, \ldots, w_n, C^{n,k}), \tag{5.11}$$

which depends on the formula (5.8) used for the integration. Corliss has shown that this process can be made adaptive with respect to order and subintervals, since the truncation error is always bounded by known intervals. ([9], pp. 150–169).

Another point in this connection which is somewhat more subtle is that real algorithms, as such, cannot usually be carried out on a computer because only floating-point numbers are available. For example, a computer program for Gaussian integration actually performs a Newton-Cotes quadrature, since the weights and nodes of the higher-order Gauss rules are not floating-point numbers. This means that the indiscriminate use of error formulas pertaining to the actual real algorithm is logically incorrect, even if accurate approximations are obtained. However, the inclusion of real algorithms by the corresponding interval algorithms is logically valid as well as computationally realizable. Since validity of results depends above all on logical correctness, inclusion methods are central to validation of numerical computation.

6. Conclusion. It appears that if a problem actually has a solution which is theoretically computable, then it should be possible to verify its existence computationally and produce an interval inclusion of it, in other words, to obtain validated results of its numerical computation. While inclusions of maximum or high quality are ideal, wider ones obtained at less expense may well be suitable for most purposes. Considerable research remains to be done to extend validation of numerical calculations from the ones which exist at present [9], [13], to other important cases.

Bibliography

1. Götz Alefeld and Jürgen Herzberger (tr. by Jon Rokne), "Introduction to Interval Computations" Academic Press, New York, 1983.
2. G. Birkhoff, "Lattice Theory" American Mathematical Society Colloquium Publications, Vol. 25, Revised Edition, American Mathematical Society, New York, 1948.
3. G. Bohlender, C. Ullrich, J. Wolff von Gudenberg, and L. B. Rall, "Pascal-SC: A Computer Language for Scientific Computation" Academic Press, Boston, 1987.

4. L. Collatz (tr. by H. Oser), "Functional Analysis and Numerical Mathematics" Academic Press, New York, 1966.

5. R. T. Gregory and D.L. Karney, "A Collection of Matrices for Testing Computational Algorithms" Wiley, New York, 1969.

6. Eldon R. Hansen (Ed.), "Topics in Interval Analysis" Oxford University Press, London, 1969.

7. S. A. Kalmykov, Yu. I. Shokin, and Z. Kh. Yuldashev, "Methods of Interval Analysis" (Russian) Science Press, Siberian Division, Novosibirsk, 1986.

8. L. V. Kantorovich, B. Z. Vulikh, and A. G. Pinsker, "Functional Analysis in Partially Ordered Spaces" (Russian) State Press for Technical-Theoretical Literature, Moscow, 1950.

9. E. Kaucher, U. Kulisch, and C. Ullrich (Eds.), "Computerarithmetic: Scientific Computation and Programming Languages" B. G. Teubner, Stuttgart, 1987.

10. E. Kaucher and W. L. Miranker, "Self-Validating Numerics for Function Space Problems" Academic Press, Orlando, 1984.

11. U. W. Kulisch (Ed.), "PASCAL-SC: A PASCAL Extension for Scientific Computation, Information Manual and Floppy Disks for IBM PC" B. G. Teubner, Stuttgart, 1987

12. U. W. Kulisch and W. L. Miranker, "Computer Arithmetic in Theory and Practice" Academic Press, New York, 1981.

13. U. W. Kulisch and W. L. Miranker (Eds.), "A New Approach to Scientific Computation" Academic Press, New York, 1983.

14. Cleve Moler, John Little, and Steve Bangert, "PC-MATLAB for MS-DOS Personal Computers, Ver. 3.2-PC" The Math Works, Inc., Sherborn, Massachusetts, 1987.

15. Ramon E. Moore, "Interval Analysis" Prentice-Hall, Englewood Cliffs, New Jersey, 1966.

16. Ramon E. Moore, "Methods and Applications of Interval Analysis" Society for Industrial and Applied Mathematics, Philadelphia, 1979.

17. Ramon E. Moore, "Computational Functional Analysis" Ellis Horwood, Chichester, 1985.

18. Ramon E. Moore (Ed.), " Reliability in Computing" Academic Press, Boston, 1988.

19. Karl L. E. Nickel (Ed.), "Interval Mathematics" Lecture Notes in Computer Science No. 29, Springer, New York, 1975.

20. Karl L. E. Nickel (Ed.), "Interval Mathematics 1980" Academic Press, New York, 1980.

21. Karl L. E. Nickel (Ed.), "Interval Mathematics 1985" Lecture Notes in Computer Science No. 212, Springer, New York, 1986.

22. Louis B. Rall (Ed.), "Error in Digital Computation, Vol. 1" Wiley, New York, 1965.

23. Louis B. Rall (Ed.), "Error in Digital Computation, Vol. 2" Wiley, New York, 1965.

24. Louis B. Rall, "Computational Solution of Nonlinear Operator Equations" Krieger, Huntington, New York, 1979.

25. Louis B. Rall, "Automatic Differentiation: Techniques and Applications" Lecture Notes in Computer Science No. 120, Springer, New York, 1981.

26. H. Ratschek and J. Rokne, "Computer Methods for the Range of Functions" Ellis Horwood, Chichester, 1984.

27. Yu. I. Shokin, "Interval Analysis" (Russian) Science Press, Siberian Division, Novosibirsk, 1981.